Efficient Software Synchronization on Large Cache Coherent Multiprocessors

Peter Magnusson psm@sics.se
Anders Landin landin@sics.se
Erik Hagersten* erik.hagersten@eng.sun.com

SICS Research Report T94:07

Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, SWEDEN

February 18, 1994

Abstract

Large-scale shared-memory multiprocessors typically have long latencies for remote data accesses. A key issue for execution performance of many common applications is the synchronization cost. The communication scalability of synchronization has been improved by the introduction of queue-based spin-locks instead of Test&(Test&Set). For architectures with long access latencies for global data, attention should also be paid to the number of global accesses that are involved in synchronization.

We present a method to characterize the performance of proposed queue lock algorithms, and apply it to previously published algorithms. We also present two new queue locks, the LH lock and the M lock. We compare the locks in terms of performance, memory requirements, code size, and required hardware support. The LH lock is the simplest of all the locks, yet requires only an atomic swap operation. The M lock is superior in terms of global accesses needed to perform synchronization and still competitive in all other criteria. We conclude that the M lock is the best overall queue lock for the class of architectures studied.

*Currently with Sun Microsystems, work done while at SICS.

1 Introduction

A major issue in designing and programing multiprocessors is the cost of synchronization. Traditionally, this area of research has focused on hardware primitives and their implementation. In recent years, however, several researchers have demonstrated that what were perceived as hardware problems could be solved in software using simple synchronization primitives as building blocks.

The difficulties involved in designing synchronization primitives vary greatly with the nature of the underlying hardware. The earliest work in the area assumed that the most powerful atomic primitive was a read or a write [Lam78, LL77]. For various reasons, we are not interested in such a restriction: to begin with, such an algorithm can never be bounded [AT92]. The inefficiency of these algorithms prompted development of more powerful hardware primitives.

These primitives, such as atomic swap, were sufficiently powerful to implement trivial locking algorithms. The common problem with these locks, such as the Test&(Test&Set) lock [SR84], is that they required $O(N^2)$ network transactions to service N processors attempting to enter a critical section simultaneously. In some cases these locks were a major contributor to network contention, including the problem of so-called "hot-spots" [PN85].

To remedy this, Goodman suggested the queue-on-sync bits [GVW89], which involves local spinning on a synchronization flag, thereby eliminating the $O(N^2)$ network operations. This solution was meant to be implemented in hardware, but Graunke and Thakkar [GT90] and Andersson [And90] implemented similar concepts in software.

The queueing principle addresses the scalability of a lock in terms of number of contending processors. Another scalability issue is performance—large shared memory architectures are today designed with some form of memory hierarchy. This implies a large cost difference between accessing local first-level caches and memory on remote nodes. This problem will be exasperated by the continuous performance improvements of microprocessors.

This report has two main contributions. First, we analyze queueing locks from the perspective of performance on a large cache-coherent shared-memory multiprocessor. Second, we present a method of analysis that is more analytical than previous approaches, allowing us to draw more general conclusions about the performance of a synchronization algorithm. Also, we demonstrate the usefulness of this approach in algorithm design by using it to guide the formulation of two new queueing spin lock algorithms.

For the duration of this report, when we refer to the scalability of an algorithm, we mean scalability in terms of machine size.

This report is organized as follows. In section 2, we present the previously best known queue locks. Then, in section 3 we describe a new queue lock, the LH lock. In section 4 we present our approach to analyzing queue lock performance, and, in section 5, we apply this method to previously published queue locks and the LH lock. We use the information from the analysis to derive a new lock, the M lock, in section 5.7. Finally, we compare all the locks and conclude in section 7.

2 Previous queue locks

In all the descriptions of a lock, atomic operations will be indicated using an extension to C. The statement "`atomic { <code> }`" will execute the statements in the code block atomically. This makes it more clear what exact semantics each lock requires. Comments are used to suggest what hardware primitive is most suitable. Furthermore, we've reformulated the locks so that variables with similar function have the same name, in order to facilitate for the reader

```
struct lock {
    int flag[0] = 1;
    int flag[1..N-1] = 0;
    int L = 0;
};

anderson_acquire(Q, P)
struct lock * Q;
int * P;
{
    atomic {              /* fetch and increment */
        *P = Q->L;
        Q->L ++;
    }
    while (Q->flag[*P % N] == 1)
        ;
    Q->flag[*P % N] = 1;
}

anderson_release(Q, P)
struct lock * Q;
int * P;
{
    Q->flag[(*P + 1) % N] = 0;
}
```

Figure 1: Andersons array-based lock

to compare them. Whenever possible, the variable `Q` will refer to the lock structure if it is composite, `L` will refer to the next item to spin upon, `P` will refer to the (local or cached) item that the processor eventually spins upon, and `I` the item that the next processor in the queue spins upon (i.e., the next processor's `P`).

2.1 Anderson's queue lock

Anderson describes an array-based lock for shared-memory multiprocessors [And90]. A similar approach was independently developed by Graunke and Thakkar (see section 2.2). An array of flags is used to allocate a unique flag for each processor that might attempt to take the lock. This flag is cached and the processor can thus read-spin locally, relieving the network. The previous processor in the queue selectively releases the lock for the next processor. A side effect of this approach is guaranteed fairness, i.e., there is no possibility of starvation. Anderson's lock requires an atomic fetch-and-increment operation.

The pseudo code for Andersons lock is given in figure 1. `%` is the modulus operator. `N` is the number of processors that can conceivably compete for the lock. As pointed out by Mellor-Crummey and Scott [MCS91b], Andersons lock is incorrect unless the number of processors is

```
struct lock {
    int flag[N];
    int L;
    int f;
}

gt_acquire(Q)
struct lock * Q;
{
    int P, f, g;
    f = Q->flag[myid];
    atomic {              /* swap */
        P = Q->L;
        g = Q->f;
        Q->L = myid;
        Q->f = f;
    }
    while (Q->flag[P] == g)
        ;
}

gt_release(Q)
struct lock * Q;
{
    Q->flag[myid] ^= 1;
}
```

Figure 2: Graunke and Thakkar's lock

an exponent of 2. This is easily corrected, and they present one such correction [MCS91b].[1]

Anderson compared his queue lock with spin locks with various back-off strategies. In all cases, Anderson's queue lock performed worse for 1-4 processors, and better for 7 or more. His implementation on the Sequent Symmetry Model B, however, did not use an atomic fetch-and-increment, since the Symmetry's atomic add instruction does not return the previous value. Instead, Anderson added an outer lock to protect the enqueue operation.

2.2 Graunke and Thakkar's queue lock

Graunke and Thakkar [GT90], independently from Anderson, developed a queue lock with similar characteristics. The pseudo code for their lock is shown in figure 2. The "`^=`" operator inverts the previous flag value, thereby releasing the lock in the present "sense." This is necessary to avoid race conditions on usage of the lock.

The `myid` value is a number from 0 to $N-1$, where N (and `N` in figure 2) is the number of processors. The lock requires only atomic swap. Whereas Anderson's lock requires a fixed

[1] Mellor-Crummey and Scott's correction consists of checking overflow and performing a global atomic add. In addition to requiring a general atomic add, the handling is done in the critical section. The version they used for performance measurement implemented a linked-list version of Anderson's lock, and thus omitted this version.

```
struct lock {
    struct lock * next;
    int flag;
};

mcs_acquire(L, I)
struct lock ** L, ** I;
{
    struct lock * before;
    *I->next = NIL;
    atomic {            /* swap */
        before = *L;
        *L = *I;
    }
    if (before != NIL) {
        I->flag = 1;
        before->next = *I;
        while (*I->flag) ;
    }
}
```

Figure 3: The MCS lock—declarations and *lock*

allocation of lock flags equal to the maximum number of processors that will ever contend for the lock, Graunke and Thakkar's lock requires a fixed allocation of lock flags equal to the number of processors regardless of their contention patterns. In other words, if it is known that at most a fixed number of processors will contend for the lock, Anderson's lock requires $O(1)$ space per lock, whereas Graunke and Thakkar's will require $O(N)$.

2.3 The MCS lock

Mellor-Crummey and Scott have described a queue-based spin lock that spins on local data, requires $O(N + P)$ space, and guarantees FIFO [MCS91a, MCS91b]. Pseudo code for the MCS lock is shown in figures 3 and 4. A global pointer `L` is maintained for each lock. If there is a queue, `L` points to a record allocated by the enqueued processor. Subsequent processors that wish to queue on the lock update the `L` pointer to point to a new record, and subsequently update the record previously pointed to by `L` to complete the link.

The compare-and-swap used by the MCS lock as described in [MCS91b] does not require the full semantics of a true compare-and-swap instruction. Instead, it is sufficient with a clear-if-equal that zeroes a memory location if the contents is equal to a parameter.

Mellor-Crummey and Scott also describe a version of the MCS lock that only requires an atomic swap. This version has the disadvantage that fairness is no longer guaranteed, and it is slightly more complex.

```
mcs_release(L, I)
struct lock ** L, ** I;
{
    int fail = 0;
    if (*I->next == NIL) {
        atomic {              /* clear_if_equal */
            if (*L == *I) {
                *L = 0;
                fail = 1;
            }
        }
        if (fail)
            return;
        while (*I->next == NIL) ;
    }
    *I->next->flag = 0;
}
```

Figure 4: The MCS lock—*release*

3 The LH lock

The main objective for the LH lock is to minimize hand-off cost. The releasing processor should not need to make any global memory accesses in order to determine where to write for the next processor to be notified. This implies that the state of the queue upon acquiring the lock must be allowed to change without affecting the hand-off code.

Once a flag has been set to indicate that the lock is free, the processor has two choices. Either synchronize (in some manner) with the next processor, or just release the lock and go away. The LH lock selects the latter route. By choosing not to synchronize, the LH algorithm can release a lock by marking a flag free and discarding the flag. This means that the release code ignores whether or not there is a queue (we obviously cannot do the same on acquire).

If there is no queue, ownership of the flag must be picked up by some other processor. In the meantime, the queue data structure must refer to the flag since it is the only item that indicates that the lock is free.

From this reasoning, the LH flag falls neatly in place. We acquire a lock by switching a flag with the lock. If the flag we receive is set, we spin on it—it will eventually be released by the processor before us in the queue. If it is not set, we're free to enter the critical section. The flag we gave to the lock is obviously set, preventing other processors from entering. When we release the lock, we clear the flag we initially put in the lock and then discard any reference to it. If a processor has tried to take the lock after us, it will be spinning on this flag. We keep the flag previously owned by the lock and can reuse it the next time we try to take a lock.

In other words, the processors and locks share a name-space of $(L + P)$ flags, where L is the number of locks and P is the number or processors. Locks each own a (free) flag initially. Processors can arrive and leave dynamically, as long as they contribute/dispose of flags in an orderly manner.

Figure 6 shows the pseudo-C code for the LH-lock. The code is "pseudo" C in the sense

```
/* local declarations */
int *I, *P;
int flag;

/* local initialization */
I = &flag;

/* global declarations */
int *L;
int L_flag;

/* global initialization */
L = &L_flag;
```

Figure 5: The simple LH-lock—declarations and initialization

that the distinction between local and global data declarations are not explicit (since they depend on what parallel extensions to C or what macro package is used). The same lock has been developed independently by Craig [Cra93].

Each lock requires an initial global pointer, `L`, that points to a global memory space for a flag. The flag is initially set to free.

The location of the pointer is statically known by all processors. In addition, each processor requires two pointers, `I` and `P`, as well as a space for a flag, for each lock that will be held. If a processor will require holding at most `n` locks at once, it will need `n` sets of `I`, `P`, and a flag. These cannot be allocated dynamically, but must be permanent.

The additional indirection for accessing these pointers will disappear if the lock is implemented as a macro or in-lined and the selection of (`I`,`P`) is known statically.[2]

Figure 7 illustrates the lock. Two processors and one global lock are involved. We will first consider only the first processor. In (a), both processors begin to execute *acquire*. The lock is initially clear. The first processor initializes it's flag (1). Then, in (b), the first processor swaps with `L`, i.e., it does `I1 -> L -> P1` (2). This processor now checks what `P1` is pointing to, and finds the lock free (3) and begins to execute the critical section.

Meanwhile, processor two initializes it's flag in (b)(4). In (c), it attempts to take the lock by performing a swap: `I2 -> L -> P2`. Inspecting what `P2` points at, it finds the lock busy, and is already enqueued. It actively spins on (6).

In (d), the processor holding the lock releases it. It writes zero to the "next" flag in the queue (7). This flag is held by `L` if the queue is empty). Next it resets `I1` to point at the flag it currently owns (8) in preparation for the next time it needs a flag. The second processor detects the clearing of the flag (7), and can enter it's critical section.

Notice the handling of flags. In essence, the flags constitute a name space that is shared between the processors' instantiations of locks and the locks themselves. Throughout figure 7, a fixed number of flags is used. However, the *ownership* of a flag changes on every acquire—for example, the first processor ends up owning the flag originally owned by the lock itself. In this way, flags are continuously reused, and only $O(N + P)$ flags are needed, where N is the

[2]The program is implemented as a procedure with C syntax, so modifying any of the parameters requires an additional indirection. In practice, the locks would preferably be implemented fully in-line.

```
lh_acquire(L, I, P)
int *L, **I, **P;
{
    **I = 1;
    atomic {  /* swap */
        *P = *L;
        *L = *I;
    }
    while (**P != 0) ;  /* spin */ ;
}

lh_release(I, P)
int **I, **P;
{
    **I = 0;
    *I = *P;
}
```

Figure 6: The LH-lock

number of locks and P is the number of processors.

To give a different view of the queueing mechanism, a longer queue of locks is shown in figure 8. Here, `I` and `P` refer to different sets of pointers in each context. The flags they point to are globally accessible. Each processor spins on the flag pointed to by its local `P`, and releases the lock by clearing the flag pointed to by its local `I`. `L` points to the last flag in the queue.

Note that there is no racing condition for the flags, as there is in Graunke and Thakkar's lock. The latter requires the flag to first be read to determine its current "sense."

4 Lock characterization

Comparing locks has previously been done either by comparing the complexity, such as determining whether acquiring a lock is $O(P)$ or $O(logP)$, by running some form of test on real hardware, or by simulation. These approaches have several limitations. A general $O()$ statement is useful mostly for theoretical scalability. In reality, this is seldom an issue—the constants involved are more important. Comparing locks on real hardware is made difficult by the absence of some primitives on suitable platforms. In some published studies, the implementation of the locks being compared bears little resemblance to the published pseudo-code.

Running a simulation is a useful approach, but as a method, simulation should be used with care. The way the simulation is designed and implemented becomes an additional set of assumptions in the analysis.

In an attempt to improve this situation, we suggest a more analytical approach.

Our primary concern is the number of global accesses of the algorithms, which is the dominant performance impediment on a large multiprocessor. At the same time we want to keep memory requirements at a minimum, as well as the overhead of the locks when there is no contention (for instance when they are superfluous).

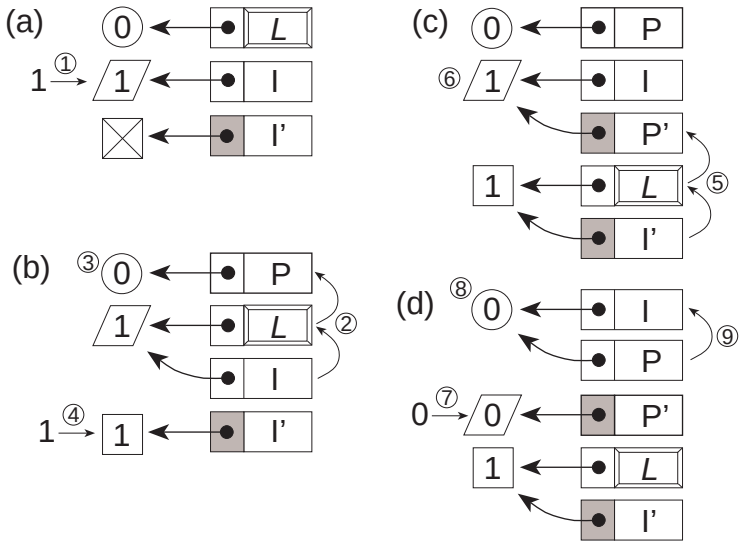

Figure 7: Example of the simple LH-lock

By hand-assembling the algorithms into a generic RISC-like assembler, and counting the number and types of operations each execution path requires, we can describe the cost of each critical path for every lock. Comparing the critical paths then allows us to make general statements about the strengths and weaknesses of the different algorithms.

Even running the test on real hardware is dubious, since the same studies seldom present accurate figures that characterize these architectures. The research community is presented with a "black box" which generates results, the validity and applicability of which cannot easily be understood.

4.1 Architectural assumptions

We assume a cache-coherent shared memory architecture, i.e., memory addresses can be read from and written to by any processor with the hardware maintaining consistency. The cache coherency protocol is presumed be of the write invalidate type, i.e., all other copies are invalidated upon writes and data is replicated in the caches upon multiple reads. Finally, we assume our memory to be sequentially consistent [Lam79]. We expect most of the results to hold for weaker memory models as well, but we do not explore the issue in this report.

Since we're interested in large machines, we assume that the principal distinction in memory access times is that between cache hits and misses. Therefore, it is the number and type of global memory accesses that will decide an algorithm's performance in relation to the size of the machine.

The interesting case for lock performance is when the locks are used frequently. This does not necessarily imply contention. Therefore, we assume that any (small) state that was copied into the cache upon the previous lock access is presumed to still reside in the cache unless another processor has written to that specific memory address (cache line). This is a *local* access. Any other access is global.

The cache-coherency protocol is assumed to handle atomic operations in a reasonable manner: if the operation involves changing the data, other copies are invalidated, and in any case a local copy is cached close to the processor. This includes Test&(Test&Set), swap, compare&swap, and clear-if-equal.

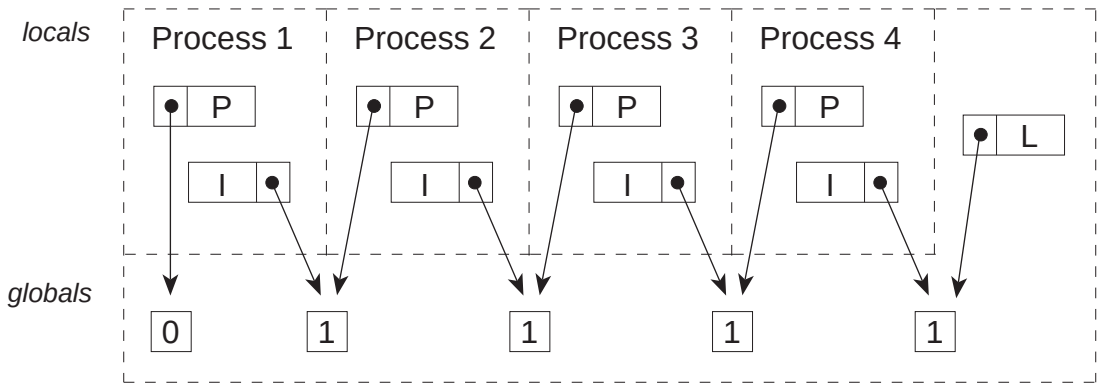

Figure 8: Enqueueing on the simple LH-lock

r_l, r_g	Time to perform a local or global read.
w_l, w_g	Time to perform a local or global write.
s_l, s_g	Time to perform a local or global swap (fetch-and-store).
a_l, a_g	Time to perform a local or global fetch and increment.
c_l, c_g	Time to perform a local or global conditional clear.
x_l, x_g	Time to perform a local or global compare and swap.
b_c	Time to execute a conditional branch that is correctly predicted. A conditional branch that is incorrectly predicted is indicated with a bar: $\overline{b}_c$. Conditional branches are presumed to be simple ones, such as branch if zero (`bz`) or branch if not zero (`bnz`).
i	Time to execute a simple instruction.

Table 1: Time characteristics of an architecture

Other than stressing the distinction between access time to the cache versus global shared memory, we do not make any assumptions about the specific relative speed of processor, cache, and shared memory.

4.2 Generic processor

Each algorithm is hand translated into "pseudo assembler", assuming a RISC-like processor. The time taken for each segment of code is divided up into generic operations: read, write, swap, execute simple instruction, and branches. A given architecture is characterized by the atomic time units in table 1. All times in the table include the time to execute the corresponding instructions. If any instructions are executed that is not a read, write, swap, or branch then this is counted as i.

The swap operation is a simple atomic read-write, returning the previous value. The compare-and-swap operation compares the old value with a parameter, and overwrites it if they are equal, returning true. Otherwise, the old value is left unchanged and the operation returns false. Clear-if-equal is similar, but always writes a zero.

The branch prediction is presumed to be statically given, i.e., all branch instructions contain

`jz rA, LB`	Jump if zero.
`inc [rA]->rB`	Atomic read and increment. Increments the value at memory address `rA`, and returns the previous value.
`mod rA,rB->rC`	`rC` = `rA` modulus `rB`.
`add rA,rB->rC`	`rC` = `rA` + `rB`.
`rA->[rB+off]`	Write `rA` to the memory location of address `rB` + `off`. `off` is known statically. `rA` can be a small constant.
`[rA+off]->rB`	Read from the memory location of address `rA` + `off`. `off` is known statically. Store the value read in `rB`.
`mask rA->rB`	A simple mask operator. A sequence of `<x>` bits starting at bit position `<y>` is copied from `rA` to the bottom of `rB`. For clarity, `<x>` and `<y>` are omitted.
`cmp rA,rB->rC`	Compare `rA` and `rB` and store a comparison vector in `rC`. Specifically, `rC` becomes zero if `rA` and `rB` are equal, and non-zero if they are different.
`xor rA,rB->rC`	Perform a logical exclusive-or operation with `rA` and `rB`, and stores the result in `rC`. `rB` can be a small constant.
`jnz rA,LB`	Jump to label `LB` if `rA` is different from zero.
`rA->[rB]->rC`	Perform a an atomic swap. The contents of memory location `rB` is copied to `rC`, and the value of `rA` is written instead. No other processor is allowed to read or write in between.
`[rA],rB eq? rC->[rA]`	Compare and swap. If the contents of memory location `rA` is equal to `rB`, then write `rC` to that location. `rC` can be a small constant, a special case being zero (compare and clear). If the swap is performed successfully, the "s" flag is set, otherwise it is cleared.
`js rA,LB`	Tests the "s" flag set by the most recent `eq?` conditional memory operation. `js` jumps to `LB` if the flag was set (i.e. jump on success).
`jns rA,LB`	Jumps if success ("s") flag is not set. See `js` above.
`j LA`	Unconditional jump to `LA`.
`jbs rA,LB`	Jump to `LB` if the bottom (lowest) bit of `rA` is set.
`shift rA->rB`	Shift `rA` (logically) right or left by a constant amount. As with the `mask` instruction, the constant is omitted for clarity.

Table 2: Generic processor instruction set

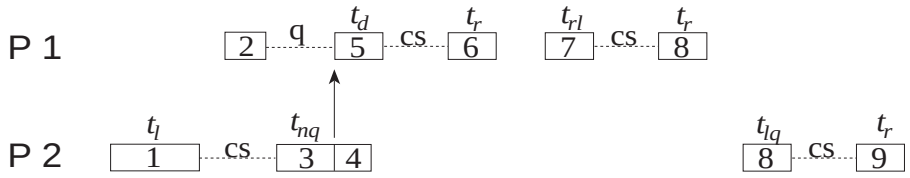

Figure 9: Timing components of a queue lock

a bit indicating whether the compiler or library code predicts the branch will be taken.

In general, some optimizations are omitted from the pseudo-C code for clarity. These locks are likely to be implemented in assembler so generic assembler-oriented optimizations are expressed in the pseudo assembler descriptions.

Table 2 describes the instructions used in the analysis.

4.3 Timing analysis

We divide the timing elements of a lock into the different execution paths. Consider figure 9. Processor 2 acquires the lock for the first time (1), spending time t_l in the lock algorithm (*lock*). It then enters its critical section. Next, processor 1 attempts to take the lock and enqueues (2). The time for this operation is not critical, since the processor has to wait anyway. Next, processor 2 completes its critical section and signals to the next processor in the queue (if any) that the lock is free (3). This takes time t_{nq} (*notice-queue*). Processor 1 now proceeds to come off the queue (5), taking time t_d to do so (*detect*). This once more is on the critical path. Processor 2, meanwhile, does any clean-up required (4) prior to proceeding with non-critical code. The time spent here is not part of the sequential component, and is furthermore reflected in the following two cases.

Processor 1 is now in its critical section. If more processors were to enter the queue, steps (3) and (5) would be repeated every time the lock was passed down the queue. We call this cost the *hand-off* cost of the queuing algorithm, $t_{ho} = t_{nq} + t_d$. When there is contention for the lock, the hand-off cost will be the part of the sequential execution that will be spent inside the lock algorithm.

In the example, processor 1 now releases the lock (6), taking time t_r (*release*). Later on, processor 1 retakes the lock (7), with no other processor having taken the lock in-between, spending time t_{rl} (*re-lock*). After its critical section, it releases the lock (8), t_r. This is the *optimistic* case, taking $t_{opt} = t_{rl} + t_r$. Since no component can be done in parallel, all the time spent in the lock is critical. The optimistic case is in a sense the cost of unnecessary locking.

Finally, processor 2 also takes (8) and releases the lock (9), but must this time spend time t_l to take the lock. This is the *pessimistic* case, $t_{pes} = t_l + t_r$. It reflects the cost of a lock that is only sporadically used.

There are two cases that we do not distinguish: taking a lock that, when we last had it, a queue formed, t_{lq}. In the figure, this would be (8). Also, the time to release the lock when there is a queue, case (4). These distinctions are not important in pratice. In [MLH94] the distinction is made.

The different times are summarized in table 3.

The total time spent in the queueing algorithm during the execution of a program is given

t_{rl}	Time to take a lock, assuming that we had the lock the previous few times. *(re-lock)*
t_l	Time to take a lock, assuming that we did not have the lock last. Last time that we did have it there was no queue. There have, however, been queues since we last held the lock. *(lock)*
t_{lq}	Time to take a lock, assuming that we did not have the lock last, and when we did last have it a queue formed. *(lock-queue)*
t_{nq}	Time to release a lock becoming noticeable when there is a queue. *(notice-queue)*
t_r	Total time to release a lock, when there is no queue. *(release)*
t_{rq}	Same as t_r, but there is a queue. *(release-queue)*
t_d	Time to come off the queue, once the previous processor has indicated that it is free. *(detect)*

Table 3: Time characteristics of queue locks

$t_{ho} = t_{nq} + t_d$	Time from a processor beginning to release a lock to the next processor in the queue acquiring that lock. *hand-off*
$t_{opt} = t_{rl} + t_r$	The optimistic time to take and release a lock, i.e., the overhead of acquire/release when the processor has held the lock the last few times (therefore no queues have formed). *optimistic*
$t_{pes} = t_l + t_r$	The slightly pessimistic time to acquire/release, i.e., the overhead of acquire/release assuming that another processor held the lock the last time, and that queues may or may not have formed in between. The previous time *this* processor held the lock, however, no queue was formed. *pessimistic*

Table 4: Summary of important critical paths in queueing locks

by:[3]

$$n_{free}(f_{rl}t_{rl} + f_l t_l + f_{lq}t_{lq} + t_{rq}) + n_{busy}(t_{nq} + t_d) \quad (1)$$

where n_{free} is the number of times some processor acquired the lock after finding it free, and n_{busy} the number of times some processor takes the lock after initially finding it busy (and thus enqueued). f_{rl}, f_l, and f_{lq} are the relative frequencies (between 0 an 1 inclusive) of locks being taken by a processor in the *re-lock*, *lock*, and *lock-queue* cases in table 3, respectively.

Idealy, we would want to find the algorithm that is best given any n_{free}, n_{busy}, f_{rl}, f_l, and f_{lq}. In our particular study, however, we're focusing on large cache-coherent multiprocessors. This means that, within reason, the number of global accesses are most important for efficiency.

The critical paths described earlier in the section are summarized in table 4.

The hand-off case is the principal bottleneck when there is contention. The goal is to minimize the overhead in "handing off" the lock to the next processor in the queue. Costs of enqueueing and dequeueing are less important in this case, since they affect only a single processor and can be done in parallel.

[3] We assume that the lock was free both before and after execution of the program.

L	The total number of locks.
P	The total number of processors.
P_q	The number of processors currently in a queue.
P_{max}	The maximum number of processors that conceivably could try for a lock simultaneously.
n	The maximum number of locks a single processor might hold at the same time.
n_{avg}	The current average number of locks held by a processor.

Table 5: Parameters affecting space requirements

The other two cases describe the critical paths when there is none or very little contention for the lock. These are the common cases for locks in general, and are a measure of the overhead taken by making the lock too sophisticated. The optimistic case ($t_{rl} + t_r$) occurs when there is no contention at all, and only a single processor is (repeatedly) taking the lock. This case highlights the calculation overhead required to figure out that the lock is still free. The pessimistic case ($t_l + t_r$) is the overhead when a lock is being (randomly) taken by processors with queues occurring only rarely. This case captures the cost of moving sufficient state to the local cache to decide that the lock is free.

4.4 Memory requirements

In the timing analysis we will also mention the space requirements for each lock. We classify memory requirements into two types: pointers and flags. Table 5 lists the parameters used to describe space requirements. Pointers are not necessarily full address space pointers—typically an offset in a data structure will be sufficient. A flag has only two states, so requires only a single bit.[4] Typically, whether or not to allocate a full cache line for a flag or a pointer (in order to reduce false sharing) will depend on their number and use. For instance, if a lock requiring $O(P)$ flags but where P_{max} is small compared to P we could pack flags with only a minor impact on performance.

We do not count usage of registers as a memory requirement.

Note that space requirements containing the parameter n, P or P_{max} require static allocation of memory.

5 Lock analysis

In this section we analyze each of the locks described previously. Each is encoded using our pseudo assembler, and the resultant timings presented.

5.1 The simple spin lock

We include the simple spin lock as a baseline. The dynamic behaviour of the spin lock is not bounded, therefore we calculate only the optimistic and pessimistic timings. We omit the pseudo C code since the algorithm is so simple.

[4] In practice its smallest size is dictated by the atomic operations available. A minimal size close to 1 bit is achievable by locking a vector of flags.

```
                                                    t_rl        t_l = t_lq    t_d
    A:  [rD + off_0] -> r1      ; fetch L           r_l         r_l
   L1:  1 -> [r1] -> r2         ; test-and-set      s_l         s_g
        jnz r2, L1              ; repeat until returns 0  b_c   b_c

                                                    t_nq = t_rq   t_r
    R:  [rD + off_0] -> r1      ; fetch L           r_l         r_l
        0 -> [r1]               ; clear the lock    w_g         w_l
```

Figure 10: Pseudo code for the simple spin lock

```
                                                    t_rl        t_l = t_lq    t_d
    A:  [rD + off_0] -> r1      ; fetch L           r_l         r_l
        inc [r1] -> r2          ; P = read_and_inc Q→L  a_l     a_g
        [rD + off_1] -> r3      ; fetch N           r_l         r_l
        mod r2, r3 -> r4        ; calc P mod N      i           i
        add rD, r4 -> r5        ; calc flag[..]     i           i
        r2 -> [rD + off_3]      ; save P            w_l         w_l
   L1:  [r5 + off_2] -> r6      ; read flag[..]     r_l (1)     r_g       r_g
        jz r6, L1               ; jump if flag[..] == 0  b_c    b_c       b_c + b̄_c/2
        0 -> [r5 + off_2]       ; write 0→flag[..]  w_l (1)     w_g       w_g

                                                    t_nq = t_rq   t_r
    R:  [rD + off_1] -> r1      ; fetch N           r_l         r_l
        [rD + off_2] -> r2      ; fetch P           r_l         r_l
        add r2, 1 -> r3         ; calc P + 1        i           i
        mod r3, r2 -> r4        ; calc (P + 1) mod N  i         i
        add rD, r4 -> r5        ; calculate flag[..]  i         i
        1 -> [r5 + off_2]       ; write 1→flag[..]  w_g         w_l
```

Figure 11: Pseudo code for Anderson's queue lock

The critical paths for the simple spin lock are:

$$t_{opt} = t_{rl} + t_r = 2r_l + s_l + b_c + w_l \quad (2)$$

$$t_{pes} = t_l + t_r = 2r_l + s_g + b_c + w_l \quad (3)$$

The case of t_{ho} is not applicable.

5.2 Anderson's queue lock

The pseudo assembler code for Anderson's lock appears in figure 11. The calculation of the array offset is spread over a few instructions. `off_2` is the fixed offset of the `flag[]` array in the data area, thus the `flag[*P % N]` is in memory location `[rD + (r2 % r3) + off_2]`.

The values labeled "(1)" in the figure are local only if the processor has acquired and released the lock at least N times, i.e., the entire `flag[]` array from figure 1 must have been cached. This implies that the optimistic case will generally be almost as bad as the pessimistic case. By nevertheless allowing the lock this benefit, we set an upper bound on the performance.

			t_{rl}	t_l	t_{lq}	t_d
A:	`[rD + off_0] -> r1`	; fetch myid	r_l	r_l	r_l	
	`[rD + off_1] -> r2`	; fetch L	r_l	r_l	r_l	
	`add rD, r1 -> r3`	; calc L→flag[myid]	i	i	i	
	`[r3 + off_2] -> r4`	; fetch d:o→f	r_l	r_l	r_g	
	`or r3, r4 -> r5`	; create myid|f	i	i	i	
	`r5 -> [r2] -> r6`	; P|g = swap(L, myid|f)	s_l	s_g	s_g	
	`mask r6 -> r7`	; extract P	i	i	i	
	`mask r6 -> r8`	; extract g	i	i	i	
	`add rD, r7 -> r9`	; calc L→flag[P]	i	i	i	
L1:	`[r9 + off_2] -> r10`	; read L→flag[P]	r_l	r_g	r_g	r_g
	`cmp r10, r8 -> r11`	; compare with g	i	i	i	$i + i/2$
	`jz r11, L1`	; branch if equal	b_c	b_c	b_c	$b_c + \overline{b}_c/2$

			$t_{nq} = t_{rq}$	t_r
R:	`[rD + off_0] -> r1`	; fetch myid	r_l	r_l
	`[rD + off_1] -> r2`	; fetch L	r_l	r_l
	`add rD, r1 -> r3`	; calc L→flag[myid]	i	i
(1)	`[r3] + off_2] -> r4`	; fetch d:o	r_l	r_l
	`xor r4, 1 -> r5`	; invert last bit	i	i
	`r5 -> [r3 + off_2]`	; write d:o	w_g	w_l

Figure 12: Pseudo code for Graunke and Thakkar's queue lock

The timing values for Anderson's lock become:

$$t_{rl} = 3r_l + 2w_l + a_l + 2i + b_c \quad (4)$$
$$t_l = 2r_l + w_l + r_g + w_g + a_g + 2i + b_c \quad (5)$$
$$t_{lq} = t_l \quad (6)$$
$$t_{nq} = 2r_l + w_g + 3i \quad (7)$$
$$t_r = 2r_l + w_l + 3i \quad (8)$$
$$t_{rq} = t_{nq} \quad (9)$$
$$t_d = r_g + w_g + b_c + \overline{b}_c/2 \quad (10)$$

And the corresponding critical paths are:

$$t_{ho} = t_{nq} + t_d = 2r_l + r_g + 2w_g + 3i + b_c + \overline{b}_c/2 \quad (11)$$
$$t_{opt} = t_{rl} + t_r = 5r_l + 3w_l + a_l + 5i + b_c \quad (12)$$
$$t_{pes} = t_l + t_r = 4r_l + 2w_l + r_g + w_g + a_g + 5i + b_c \quad (13)$$

Anderson's lock requires LP_{max} flags and $L + nP_q$ pointers.

5.3 Graunke and Thakkar's queue lock

Figure 12 contains the pseudo assembler code for Graunke and Thakkar's queue lock. `L` and `f` in the lock structure (see figure 2) are stored in a single 32-bit word and can thus be swapped atomically.

			t_{rl}	t_l	t_{lq}
A:	`[rD + off_0] -> r1`	I	r_l	r_l	r_l
	`0 -> [r1 + off_1]`	I→next = 0	w_l	w_l	w_g
	`[rD + off_2] -> r2`	L	r_l	r_l	r_l
(1)	`r1 -> [r2] -> r3`	P = swap(L, I)	s_l	s_g	s_g
	`jz r3, L2`	if (P != 0)	b_c	b_c	b_c
	`1 -> [r1 + off_3]`	I→locked = 1			
	`I -> [r3 + off_4]`	P→next = 1			(t_d)
L1:	`[r1 + off_3] -> r4`	I→locked			r_g
	`jnz r4, L1`	while (I→locked)			$b_c + \overline{b}_c/2$
L2:					
			$t_{nq} = t_{rq}$		t_r
R:	`[rD + off_0] -> r1`	I	r_l		r_l
	`[r1 + off_1] -> r2`	I→next	r_g		r_l
	`jnz r2, L2`	if (I→next != 0)	b_c		$\overline{b}_c$
	`[rD + off_2] -> r3`	L			r_l
(2)	`[r3],r2 eq? 0->[r3]`	clear_if_equal(L,I)			c_l
	`js L3`	jump if cleared			b_c
L1:	`[r1 + off_1] -> r2`	I→next			
	`bz r2,L1`	while (I→next == 0)			
L2:	`0 -> [r2 + off_3]`	I→next→locked = 0	w_g		
L3:					

Figure 13: Pseudo code for the MCS lock with clear_if_equal()

The timing values in line (1) of figure 12 are local since, if there's a queue, this value has only been read by other processors, not written.

The timing values for Graunke and Thakkar's lock become:

$$t_{rl} = 4r_l + s_l + 6i + b_c \quad (14)$$

$$t_l = 3r_l + r_g + s_g + 6i + b_c \quad (15)$$

$$t_{lq} = 2r_l + 2r_g + s_g + 6i + b_c \quad (16)$$

$$t_{nq} = 3r_l + w_g + 2i \quad (17)$$

$$t_r = 3r_l + w_l + 2i \quad (18)$$

$$t_{rq} = t_{nq} \quad (19)$$

$$t_d = r_g + i + i/2 + b_c + \overline{b}_c/2 \quad (20)$$

And the corresponding critical paths are:

$$t_{ho} = t_{nq} + t_d = 3r_l + r_g + w_g + 3\frac{1}{2}i + b_c + \overline{b}_c/2 \quad (21)$$

$$t_{opt} = t_{rl} + t_r = 7r_l + w_l + s_l + 8i + b_c \quad (22)$$

$$t_{pes} = t_l + t_r = 6r_l + w_l + r_g + s_g + 8i + b_c \quad (23)$$

Graunke and Thakkar's lock requires $L(P+1) + P_q$ flags and $L(P+1) + P_q$ pointers.

5.4 The MCS lock

```
                                                        t_n   t_nq  t_r
R:   [rD + off_0] -> r1   ; I                           r_l   r_l   r_l
     [r1 + off_1] -> r2   ; I→next                      r_l   r_g   r_l
     bnz r2, L1           ; if (I→next != 0)            b̄_c   b_c   b̄_c
     [rD + off_2] -> r3   ; L                           r_l         r_l
     0 -> [r3] -> r4      ; r4 = swap(L, 0)             s_l         s_l
     cmp r1, r4           ; compare L, I                            i
     bz r4,L2             ; jump if equal                           b_c
     :
     :
L1:  0 -> [r2 + off_3]    ; I→next→locked = 0                 w_g
L2:
```

Figure 14: Pseudo code for MCS_release() with swap()

Figure 13 shows the pseudo assembler for the MCS lock with such an instruction.

If the lock has been acquired by other processors, the global pointer `L` will be located remotely, requiring a global operation to fetch. `I->next` will be local, however, unless it was involved in a queue the last time. In this case, `I->next` will have been accessed by a remote processor to indicate release target, and will thus be replicated on that node. Similar distinctions apply for release. All in all, the MCS lock has a less complex cache behaviour than the LH-lock.

Upon releasing a lock, the MCS lock is forced to take a branch. Unfortunately, both targets are common but under different circumstances, forcing the MCS lock to make a poor static prediction. We've chosen to prioritize lock release when there is a queue, since this affects other processors.

The timing figures noted in figure 13 are summarized as follows:

$$t_{rl} = 2r_l + w_l + s_l + b_c \quad (24)$$

$$t_l = 2r_l + w_l + s_g + b_c \quad (25)$$

$$t_{lq} = 2r_l + w_g + s_g + b_c \quad (26)$$

$$t_{nq} = r_l + r_g + w_g + b_c \quad (27)$$

$$t_r = 3r_l + c_l + b_c + \overline{b}_c \quad (28)$$

$$t_{rq} = t_{nq} \quad (29)$$

$$t_d = r_g + b_c + \overline{b}_c/2 \quad (30)$$

And the corresponding critical paths for the MCS lock are:

$$t_{ho} = t_{nq} + t_d = r_l + 2r_g + w_g + 2b_c + \overline{b}_c/2 \quad (31)$$

$$t_{opt} = t_{rl} + t_r = 5r_l + w_l + c_l + s_l + 2b_c + \overline{b}_c \quad (32)$$

$$t_{pes} = t_l + t_r = 5r_l + w_l + c_l + s_g + 2b_c + \overline{b}_c \quad (33)$$

Notice that the clear-if-equal instruction is done only locally in any critical path.

The version of the MCS lock that uses only the swap primitive, though more complex, has very similar critical paths—see figure 14. In equations 32 and 33 above, c_l is replaced with $s_l + i$. Equation 31 is unaffected. The complicated cases occur when a processor that releases a lock erroneously detects that the queue is empty. In other words, no additional global operations are induced in the critical paths.

The memory requirements of the MCS lock is $P_q + n_{avg}P$ flags and $L + P_q + n_{avg}P$ pointers.

```
                                                  t_rl   t_l = t_lq      t_d
A:  [rL + off_0] -> r1   ; I                      r_l    r_l
    [rG + off_2] -> r2   ; L                      r_l    r_l
    1 -> [r1]            ; *I = 1                 w_l    w_g
    r1 -> [r3] -> r2     ; P = swap(L, I)         s_l    s_g
    r2 -> [rL + off_1]   ; save P                 w_l    w_l
L1: [r2] -> r3           ; while (*P != 0)        r_l    r_g      r_g
    bz r3, L1                                     b_c    b_c      b_c + b̄_c/2

                                                  t_nq   t_r      t_rq
R:  [rL + off_0] -> r1   ; I                      r_l    r_l      r_l
    0 -> [r1]            ; *I = 0                 w_g    w_l      w_g
    [rL + off_1] -> r2   ; P                             r_l      r_l
    r2 -> [rL + off_0]   ; I = P                         w_l      w_l
```

Figure 15: Pseudo code for the LH lock

5.5 The LH lock

Generic assembler code for the LH lock is given in figure 15. The code is shorter than typical entry/exit code for procedure calls, so it is presumed to be in-line. Therefore, the extra indirection to obtain the pointer locations is optimized away. The timing values indicated contribute to the different execution paths. The `A` and `R` labels indicate entry points for the acquire and release routines, respectively. `off_x` indicates a statically known offset into local or global data. The notation "`[r1]`" refers to the memory at the address contained in register `r1`.

The pointers `I`, `P`, and `L` are assumed to be at fixed offsets from pre-initialized registers (`rL` and `rG` for local and global data areas, respectively). A simple load/store architecture is presumed, thus memory-to-memory copying must go through a register.

The cache behaviour of the LH lock is curious. Upon entry to a lock, the locations of the flags owned by the processor and the lock will depend upon their usage in the previous two locks. Consider figure 16. The initial state is the worst case, i.e., there has been previous contention for the lock. The flag owned by the lock is located remotely, and the flag owned by the processor is replicated remotely (at the processor that held the lock immediately before we last held it).

We now assume no further contention. After the first acquire/release of the lock, the flag previously owned by the lock is now replicated locally, and the flag owned by the lock resides locally. After the second acquire/release, both flags reside locally. This behaviour is caused by the single write performed by the algorithm. The flag `L` itself is migrated locally after the first acquire/release.

If, after (1) in figure 16, the lock is acquired by another processor, both `L` and `B` will migrate. Thus, the next time around the processor will be back in the worst-case initial state. Hence, t_l takes the same penalties as t_{lq} does.

In summary, we have the following timings for the LH lock:

$$t_{rl} = 3r_l + 2w_l + s_l + b_c \tag{34}$$

$$t_l = 2r_l + w_l + r_g + w_g + s_g + b_c \tag{35}$$

$$t_{lq} = t_l \tag{36}$$

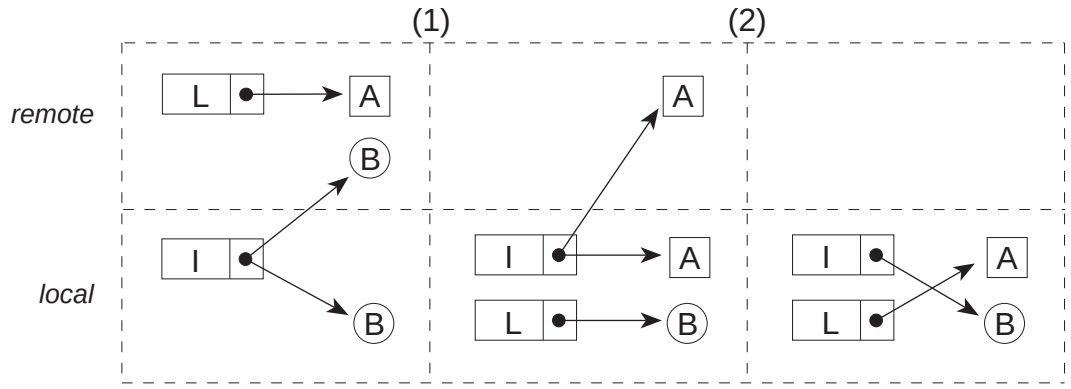

Figure 16: Cache behaviour of the LH lock

$$t_{nq} = r_l + w_g \quad (37)$$
$$t_r = 2r_l + 2w_l \quad (38)$$
$$t_{rq} = 2r_l + w_l + w_g \quad (39)$$
$$t_d = r_g + b_c + \overline{b}_c/2 \quad (40)$$

The value $b_c/2$ in equation 40 indicates that this value is the worst case cost if the update to the global variable being spun upon is just missed. This value comes from a possibly unnecessary conditional branch in the *while* in figure 15 . On average, half of this cost will be payed every time the enqueued processor is forced to spin.

The critical paths from table 4 for the LH lock therefore become:

$$t_{ho} = t_{nq} + t_d = r_l + w_g + r_g + b_c + \overline{b}_c/2 \quad (41)$$
$$t_{opt} = t_{rl} + t_r = 5r_l + 4w_l + s_l + b_c \quad (42)$$
$$t_{pes} = t_l + t_r = 4r_l + 3w_l + r_g + w_g + s_g + b_c \quad (43)$$

Notice that the control flow is lean: the only branch that is not correctly statically predicted is the one involved in spinning on a local variable. Since it's used for a loop, it will always execute both paths.

5.6 Comparing the MCS lock with the LH lock

So far we have made no assumptions about the specific values of the timing delays due to the architecture, nor their relevant frequency. In this section we will compare the lh-lock with the MCS lock while keeping these parameters either undefined or as general as possible.

To begin with, we form the differences between the MCS times and the LH-lock times, by subtracting the latter from the former. Thus, for instance, $\Delta t_{rl} = t_{rl}(\text{MCS}) - t_{rl}(\text{LH-lock})$.

$$\Delta t_{rl} = -r_l - w_l \quad (44)$$
$$\Delta t_l = -r_g - w_g \quad (45)$$
$$\Delta t_{lq} = -w_l - r_g \quad (46)$$
$$\Delta t_n = 2r_l - w_l + c_l + \overline{b}_c \quad (47)$$
$$\Delta t_{nq} = r_g + b_c \quad (48)$$
$$\Delta t_r = r_l + c_l - 2w_l + \overline{b}_c + b_c \quad (49)$$

$$\Delta t_{rq} = r_g - r_l - w_l + b_c \quad (50)$$
$$\Delta t_d = 0 \quad (51)$$

From the perspective of scalability, it is the global references that is of interest. If we set all other timings to zero, we get:

$$\Delta t_{rl} = \Delta t_n = \Delta t_r = \Delta t_d = 0 \quad (52)$$
$$\Delta t_l = -w_g - r_g \quad (53)$$
$$\Delta t_{lq} = -r_g \quad (54)$$
$$\Delta t_{nq} = r_g \quad (55)$$
$$\Delta t_{rq} = r_g \quad (56)$$

In other words, the LH-lock requires an additional global read and write to acquire the lock unless the same processor held the lock the previous two times. The MCS lock, in turn, requires an additional global read to release a lock if there is a queue.

At first sight, this might look like an almost even tradeoff. When there is contention, the LH-lock moves a global read from release to acquire, and can eliminate it if it has held the lock at least twice in a row. This is not a very interesting situation, since if there is contention, it is not likely that the processor will re-acquire the lock multiple times.

However, acquiring a lock can be done when the critical section is busy, which is no loss since the alternative is to spin. Releasing the lock when there is a queue is the bottleneck: the additional global read will slow down *all* the waiting processors.

This becomes more clear if we construct the delta values for the critical paths in table 4:

$$\Delta(t_{ho}) = r_g + b_c \quad (57)$$
$$\Delta(t_{opt}) = c_l - 3w_l + b_c + \overline{b}_c \quad (58)$$
$$\Delta(t_{pes}) = r_l + c_l - 2w_l - r_g - w_g + b_c + \overline{b}_c \quad (59)$$

It is equation 57 that is important when there is a high degree of contention. The time taken to add a processor to the queue or clean up after removal can be overlapped with other useful work. The MCS lock is not only slower in handing off, but requires a global operation.

In the case of low contention, the effect of hand-off timing will be less important. Depending on the application, performance will be determined by the mix of optimistic and pessimistic accesses to the lock.

For the optimistic lock, the locks perform approximately the same. The pessimistic lock is a more likely case: multiple processors randomly taking the lock but not holding on to it long enough to create queues. Here, the poor cache behaviour of the LH-lock causes two global operations. On smaller machines, the locks will be roughly equivalent, but the LH-lock's behaviour will scale poorly.

Unfortunately, one of the primary objectives in searching for a fast queue lock is that it should work well in the common case where there is none or very little contention. The reason is that this would encourage programmers to always use the same lock, relieving the programmer of the effort of deciding whether or not there will be contention for a particular lock.

Finally, notice that the lh-lock requires 10 lines of generic assembler, whereas the MCS lock requires 18 lines.

The issue of implementing a conditional version of the LH lock is beyond the scope of this report.

5.7 The M lock

In this section we will apply our analytical approach in a reversed manner. We will use the details of timing to tell us exactly what is "wrong" with the LH lock algorithm, and the critical paths to constrain us to look only for algorithms that are strictly better.

To improve the LH lock we need to understand why it requires an additional two global accesses in the pessimistic case. The answer has already been given in figure 16. The aggressive method of enqueueing spoils the cache behaviour. The processors exchange ownership of flags on each acquire/release, therefore forcing an unnecessary global read/write on each exchange.

If the processor could know upon acquire that there is no queue, then a global read could be avoided. Similarly, if the processor knew that there was no queue upon release, it would not be necessary to switch flags with the lock. This would eliminate the the global write when `*I` is written to in the next use of the lock.[5]

The idea is that if, upon releasing the lock, no queue has formed, then we do not need to exchange a flag with the lock. If we can avoid the switch, we will retain an already localized flag for the next acquire. However, this manipulation must be done off the critical path. In particular, it must not delay the hand-off time. The reason is that if there's a queue, `L` will no longer be local, and hence we will be inducing a global access in the critical path. This is in fact one of the problems with the MCS lock, where an "optimization" is to read the local pointer that, if a queue has formed, will no longer be local (see figures 3 and 4). Therefore, we wish to keep the core advantage of the LH lock, namely, that only a single global write is needed to notify that the lock is free.

In order to ascertain that the queue has indeed not changed, it is not sufficient to compare `*L` with `*I`. The reason is that once we've released the flag pointed to by `*I`, we're creating a racing condition for the value of `*L`. Another processor could acquire the lock, take over `*I`, release the lock, and re-acquire it, thereby again setting `*L` and `*I` equal, but in a new context.

Therefore, we must uniquely identify the processor that last modified `*L`. We do this by merging two values into `*L`—the pointer to a flag, and the id of the processor that last changed the value. Only if this value has changed do we write back the old pointer value. This is easily arranged with 64-bit or even 32-bit integers.

Figures 17 and 18 shows the first version of the M lock. `myid` is presumed to contain the number of the processor (numbered 1 and up). `*Q` points to a structure containing a pointer to a flag and an id. By having the default id as zero, a processor can check if the lock is busy by checking the `old_id` field. This removes the superfluous (potentially global) indirection of `*P`. Upon release, if there is no queue, the algorithm reuses the old (locally cached) flag. This removes a future global write when `**I` is initialized in `m_acquire()`.

Figures 19 and 20 shows the assembler code and timing for the first version of the M lock. The `*I` and `id` values are stored in a single merged integer (32 or 64 bits). The bottom bit of this integer is always set. The instruction `jbs` jumps on bottom bit set. This is faster on many processors (such as the Alpha and 881xx), and makes the algorithm independent of numbering scheme.

We form only the critical paths for this version of the M lock:

$$t_{ho} = t_{nq} + t_d = r_l + r_g + w_g + i + b_c + \overline{b}_c/2 \tag{60}$$

$$t_{opt} = t_{rl} + t_r = 5r_l + 3w_l + s_l + x_l + 2i + 2b_c \tag{61}$$

[5] We initially solved this problem by making use of a more general fetch-and-Φ operation. A further refinement of the algorithm reduced it to requiring simpler atomic operations. We concluded that it can be fruitful to drop the restrictions of available atomic operations early on in designing an algorithm, and wait until one or more alternatives crystalize before attempting to implement them using more restrictive operations.

```
struct lock {
          int *L;
          int id;
}

m_acquire_1(Q, I, P, old_id)
struct lock *Q;
int **I, **P, *old_id;
{
          **I = 1;
          atomic {              /* swap */
              *P = Q->L;
              *old_id = Q->id;
              Q->L = *I;
              Q->id = myid;
          }
          if (*old_id > 0)
              while (**P != 0) /* spin */ ;
}
```

Figure 17: A first version of the `m_lock()`

	Hand-off ($t_{nq} + t_d$)	Pessimistic ($t_l + t_r$)
Anderson	$r_g + 2w_g$	$r_g + w_g + s_g$
Graunke/Thakkar	$r_g + w_g$	$r_g + s_g$
Mellor-Crummey/Scott	$2r_g + w_g$	s_g
LH	$r_g + w_g$	$r_g + w_g + s_g$
first M	$r_g + w_g$	s_g

Table 6: Global accesses for different locks. Note that none of the algorithms require global accesses in the optimistic case.

$$t_{pes} = t_l + t_r = 5r_l + 3w_l + x_l + s_g + 2i + 2b_c \tag{62}$$

We've eliminated the superfluous global operations in the pessimistic case. This lock is now strictly more scalable than any of the locks analyzed in section 5. This is clear from table 6. It is the only lock that has the lowest number of global accesses in both critical paths. Ergo, it will be equal to or better than any linear combination of the critical paths.

Three problems remain with this version. First, it retains the unpleasant characteristic from the LH lock of requiring an extra flag per lock. Second, it requires more local operations than the other queue locks. Let us compare it with the MCS lock, for instance:

$$\Delta(t_{ho})_{MCS-M'} = r_g + b_c - i \tag{63}$$
$$\Delta(t_{opt})_{MCS-M'} = -2w_l + \overline{b}_c - i - b_c \tag{64}$$
$$\Delta(t_{pes})_{MCS-M'} = -2w_l + \overline{b}_c - i - b_c \tag{65}$$

The non-memory operations are approximately better in the M' lock (a misspredicted branch

```
m_release_1(Q, I, P, old_id)
struct lock *Q;
**I, **P, *old_id;
{
          int failed = 0;
          **I = 0;
          atomic {            /* compare-and-swap */
              if (Q->id == myid) {
                  Q->L = *P;
                  Q->id = *old_id;
              } else
                  failed = 1;
          }
          if (failed)
              *I = *P;
}
```

Figure 18: A first version of `m_release()`

will typically have at least one cycle overhead). However, the M' lock has two additional local memory operations in the optimistic and pessimistic cases.

This additional cost is obviously not relevant from a scalability standpoint. It does, however, deter from using the lock in ordinary situations when the programmer is uncertain about the contention behaviour.

Finally, the atomic operation required is a full *compare-and-swap*, as opposed to the *compare-and-clear* used by the MCS lock. This is clearly a disadvantage, since the latter is easily implemented with the former but not vice-versa.

Fortunately, we can solve the memory overhead, reduce the local work, and lower the atomic operation requirement to a *compare-and-clear*, all at the same time.

The memory overhead is caused by `Q` having to own a flag even when not in use. Now, in the optimistic and pessimistic cases, when we acquire/release this lock the flag pointed to is never used (we reuse `*I` every time). Furthermore, we're saving and restoring this pointer each time, causing an unnecessary r_l/w_l pair for both of these critical paths.

Let us discard the lock's flag. Now, if upon release of the lock we detect a queue, then we cannot reuse our flag (as discussed previously). This means that we would have to arbitrate with the next processor to regain our flag. However, this implies further synchronization which impacts on the next processor's critical section. Alternatively, we would have to wait for it to release the lock in its turn so as not to impact the hand-off chain.

Instead, let us in this case allocate a flag using a new procedure, `alloc_flag()`. Similarly, if we receive a `P` to spin upon, we subsequently discard this flag with `free_flag()`. These procedures can be made very efficient. First, the size of allocation is fixed, and very few will be necessary. This allows us to have a local "cache" of flags, and only extremely rarely will we be forced to perform global shuffling of flags. Second, each processor will have at most one flag outside the acquire/release section. Thus, alloc/free can be implemented in a heap fashion.

This approach introduces a new problem. We do not want to have alloc/free code in any of our critical paths. Even if they are efficient, they will counteract our goal of reducing local

			t_{rl}	t_l	t_r	t_d
A:	`[rD + off_0] -> r1`	; fetch I|id	r_l	r_l	r_l	
	`shift r1 -> r2`	; extract I	i	i	i	
	`1 -> [r2]`	; *I = 1	w_l	w_l	w_g	
	`[rD + off_1] -> r3`	; fetch Q	r_l	r_l	r_l	
	`r1 -> [r3] -> r4`	; I|id|1→[L]→P|old_id|f	s_l	s_g	s_g	
	`r4 -> [rD + off_2]`	; save P|old_id|f	w_l	w_l	w_l	
	`jbs r4, L2`	; jmp if (f == 0)	b_c	b_c	b_c	
	`shift r4 -> r5`	; extract P				
L1:	`[r5] -> r6`	; read *P				r_g
	`jnz r6, L1`	; jmp if (*P == 1)				$b_c + \overline{b}_c/2$
L2:						

Figure 19: Pseudo code and timing for first version of `m_acquire()`

			t_{nq}	t_r	t_{rq}
R:	`[rD + off_0] -> r1`	; fetch I|id|1	r_l	r_l	r_l
	`shift r1 -> r2`	; extract I	i	i	i
	`0 -> [r2]`	; *I = 0	w_g	w_l	w_g
	`[rD + off_1] -> r3`	; fetch L		r_l	r_l
	`[rD + off_2] -> r4`	; fetch P|old_id		r_l	r_l
	`[r3],r1 eq?  r4 -> [r3]`	; if (*Q == I|id|1) *Q=P|old_id|f		x_l	x_g
	`js L1`	; jmp if (success)		b_c	$\overline{b}_c$
	`mask r4 -> r5`	; extract P			i
	`mask r4 -> r6`	; extract id|1			i
	`or r5,r6 -> r7`	; form P|id|1			i
	`r7 -> [rD + off_2]`	; save P|id|1			w_l
L1:					

Figure 20: Pseudo code and timing for first version of `m_release()`

work. If we're about to spin on a flag (i.e., the processor is enqueueing), then we know that we will want to discard `*P`. The reason comes from our earlier discussion—the alternative would be a further synchronization with the previous processor in the queue. But we do not want to call `free_flag(*P)` after `*P` has been used since this would put the code in the hand-off critical path. The problem is that this creates a race condition for `*P`. We could create a `prepare_free_flag()` procedure, which would effectivate freeing `*P` after it has been set to 0. However, this would also create a race condition.

Finally, we could associate `*I` with `*P`, so that `*P` is released only after `*P` is 0 *and* `*I` is 0. This would increase memory requirements, since `*P` might never be released. Also, this presupposes a complex algorithm. The real dilemma, however, is that this assumes that some other processor (one that is spinning anyway, for instance) polling `*I`. Since the latter is meant to be localized, polling it would introduce a w_g cost by the `**I = 1` statement in acquire. The processor that released `*P` cannot poll, since this would imply adding code to the optimistic and pessimistic critical paths.

The solution comes from considering this last statement. If a processor has to "poll" some data structure in order to release it, how would it do this outside a critical path? Well, simply wait until the next time we try to acquire a lock and we're forced to spin! In other words,

```
m_acquire(Q, I, K)
struct lock *Q;
int **I, **K;
{
    int old_id;
    **I = 1;
    atomic {        /* swap */
        *P = Q->L;
        old_id = Q->id;
        Q->L = *I;
        Q->id = myid;
    }
    if (old_id > 0)
        if (*K)
            free_flag(*K);
        *K = *P;
        while (**P != 0) /* spin */ ;
}
```

Figure 21: A final version of `m_acquire()`

the processor should "trail" a `free_flag()` operation until the next acquire. This trailing `*P` could of course be reused instead of calling `alloc_flag()`, so this has the added benefit of caching a spare flag. This adds memory overhead with P flags, but on the other hand reduces it with L flags (this is the cost of retaing the reduced memory requirement of the first version of the M lock). The number of pointers is also increased with P.

The code for the final version of the M lock is shown in figures 21 and 22. The trailing `*K` maintains an old `*P` value. Notice that `free_flag()` is called when the processor is about to spin anyway, and `alloc_flag()` is called only when a queue has formed. This way, both segments of code are outside the critical paths. As with the previous version, myid is numbered 1 and up.

The reduced memory overhead is achieved by not having to save `old_id` and `*P` after acquire.

Also, notice that the atomic operation is now compare-and-clear.

If a queue forms, the processor that began the queue "volunteers" a flag. Upon leaving the queue, it may or may not still have a flag in `*K`. If not, it needs to allocate one. Processors that join an existing queue, however, simply replace `*I` with an old `*P` *after* leaving the queue.

The pseudo assembler code and timing analysis is given in figures 23 and 24. The code is very similar to the first version of the M lock, except that some local work has disappeared from the critical paths and t_{rq} has become more complex. The timings become:

$$t_{rl} = 2r_l + w_l + s_l + i + b_c \quad (66)$$

$$t_l = 2r_l + w_l + s_g + i + b_c \quad (67)$$

$$t_{lq} = 2r_l + w_g + s_g + i + b_c \quad (68)$$

$$t_{nq} = r_l + w_g + i \quad (69)$$

$$t_r = 2r_l + w_l + c_l + i + b_c \quad (70)$$

```
m_release(Q, I, K)
struct lock *Q;
int **I, **K;
{
    int failed = 0;
    **I = 0;
    atomic {        /* compare-and-clear */
        if (Q->id == myid) {
            Q->id = 0;
        } else
            failed = 1;
    }
    if (failed)
        if (*K) {
            *I = *K;
            *K = 0;
        } else
            *I = alloc_flag();
}
```

Figure 22: A final version of `m_release()`

$$t_d = r_g + b_c + \overline{b}_c/2 \tag{71}$$

The critical paths of the final M lock are:

$$t_{ho} = t_{nq} + t_d = r_l + r_g + w_g + i + b_c + \overline{b}_c/2 \tag{72}$$
$$t_{opt} = t_{rl} + t_r = 4r_l + 2w_l + s_l + c_l + 2i + 2b_c \tag{73}$$
$$t_{pes} = t_l + t_r = 4r_l + 2w_l + c_l + s_g + 2i + 2b_c \tag{74}$$

And if we now form the delta with the MCS lock we get:

$$\Delta(t_{ho})_{MCS-M'} = r_g + b_c - i \tag{75}$$
$$\Delta(t_{opt})_{MCS-M'} = r_l - w_l + \overline{b}_c - i - b_c \tag{76}$$
$$\Delta(t_{pes})_{MCS-M'} = r_l - w_l + \overline{b}_c - i - b_c \tag{77}$$

In other words, the M lock performs almost identically in the optimistic and pessimistic cases, but one global read better in hand off. The improvement over the first M lock is thus $r_l + w_l$ (in addition, of course, to the memory and atomic operation improvements). It still nearly retains the minimal cost of hand-off from the LH lock.

6 Comparison of locks

Table 7 summarizes the critical paths for all the locks we've looked at in this report. The "baseline" entry contains the minimum of all the studied locks, and has been subtracted from all the locks to facilitate comparison. In table 8, we include only the global accesses from

			t_{rl}	t_l	t_r	t_d
A:	`[rD + off_0] -> r1`	; fetch I|id	r_l	r_l	r_l	
	`shift r1 -> r2`	; extract I	i	i	i	
	`1 -> [r2]`	; *I = 1	w_l	w_l	w_g	
	`[rD + off_1] -> r3`	; fetch Q	r_l	r_l	r_l	
	`r1 -> [r3] -> r4`	; I|id→[Q]→P|old_id	s_l	s_g	s_g	
	`jbs r4, L2`	; jmp if (old_id == 0)	b_c	b_c	b_c	
	`[rD + off_3] -> r7`	; fetch K				
	`.....`	; if (K!=0) free_flag()				
	`shift r4 -> r5`	; extract P				
	`r5 -> [rD + off_3]`	; K = P				
L1:	`[r5] -> r6`	; read *P				r_g
	`jnz r6, L1`	; jmp if (*P == 1)				$b_c + \overline{b}_c/2$
L2:						

Figure 23: Pseudo code and timing for `m_acquire()`

			t_{nq}	t_r	t_{rq}
R:	`[rD + off_0] -> r1`	; fetch I|id	r_l	r_l	r_l
	`shift r1 -> r2`	; extract I	i	i	i
	`0 -> [r2]`	; *I = 0	w_g	w_l	w_g
	`[rD + off_1] -> r3`	; fetch Q		r_l	r_l
	`[r3],r1 eq?  r4 -> [r3]`	; if (*Q == I|id) *Q=0		c_l	c_g
	`js L2`	; jmp if (success)		b_c	$\overline{b}_c$
	`[rD + off_3] -> r8`	; fetch K			r_l
	`jz r8, L1`	; jmp if (K == 0)			:
	`r8 -> [r2]`	; I = K			:
	`0 -> [rD + off_3]`	; K = 0			:
	`j L2`	; goto L2			:
L1:	`.....`	; I = alloc_flag()			:
L2:					

Figure 24: Pseudo code and timing for `m_release()`

table 7, together with other important characteristics of each lock. As the tables indicate, we have six criteria: performance for the three critical paths, size of code, atomic operation requirements, and memory requirements. In this discussion, we consider only the version of the MCS lock that is fair: this is reasonable, since we are searching for a general lock (i.e., library implemention).

Two of the critical paths, hand off and pessimistic, are easy to draw conclusions from. For large multiprocessors, global memory accesses will be considerably more expensive than any of the other operations, and so we need only consult the first two rows of table 8. Here, the M lock is superior or equivalent to all the alternatives.

The third critical path, the optimistic case, is more dependent on architectural details. If, as an example, we assume the following relative costs:

$$\overline{b}_c = 2b_c = 2i = 2w_l = 2r_l = s_l = c_l = a_l \tag{78}$$

then we arrive at the values in the third row in table 8. The simple design of the LH lock pays off, but the MCS and M locks are only 16% slower.

	Hand-off $(t_{nq} + t_d)$	Optimistic $(t_{rl} + t_r)$	Pessimistic $(t_l + t_r)$
Anderson	$r_l + w_g + 3i$	$2w_l + r_l + a_l + 5i$	$r_g + w_g + a_g + 2w_l + 5i$
GT	$2r_l + 3\frac{1}{2}i$	$3r_l + s_l + 8i$	$2r_l + w_l + r_g + s_g + 8i + b_c$
MCS	$r_g + b_c$	$r_l + s_l + c_l + b_c + \overline{b}_c$	$r_l + w_l + s_g + c_l + 2b_c + \overline{b}_c$
LH	0	$r_l + 3w_l + s_l$	$3w_l + w_g + s_g + r_g + b_c$
first M	i	$r_l + 2w_l + s_l + x_l + 2i + b_c$	$r_l + 3w_l + x_l + s_g + 2i + 2b_c$
final M	i	$w_l + s_l + c_l + 2i + b_c$	$2w_l + c_l + s_g + 2i + 2b_c$
Baseline	$r_l + r_g + w_g + b_c + \overline{b}_c/2$	$4r_l + w_l + b_c$	$4r_l$

Table 7: Summary of queue lock performance for critical paths

	Anderson	MCS	GT	LH	M
Hand-off $(t_{nq} + t_d)$	$r_g + 2w_g$	$2r_g + w_g$	$r_g + w_g$	$r_g + w_g$	$r_g + w_g$
Pessimistic $(t_l + t_r)$	$r_g + w_g + a_g$	s_g	$r_g + s_g$	$r_g + w_g + s_g$	s_g
Optimistic $(t_{rl} + t_r)$	16	14	19	12	14
Lines of code	15	18	18	11	24+
Atomic operations	fetch_and_inc	c_&_c (swap)	swap	swap	swap + c_&_c
Number of flags	LP_{max}	$P_q + n_{avg}P$	$L(P+1) + P_q$	$L + nP$	$(n+1)P$
Number of pointers	$L + P_q + n_{avg}P$	$L + P_q + n_{avg}P$	$L(P+1) + P_q$	$L + nP$	$(n+1)P + P_q + L$

Table 8: Queue lock comparison

The code size is clearly in favour of the LH lock. The 24+ entry for the M lock means that, in addition to 24 lines of code, there are 2 procedure calls. This is more than twice as much as the LH lock, with the MCS and GT locks in-between. The smaller the code, the less of an impact for in-lining the locks.

The compare-and-clear can be just as easily implemented as atomic swap on current architectures, such as the R4000 and Alpha [MIP91, Dig92]. Hence we are not too concerned with the atomic operation requirements. Nevertheless, if only an atomic swap is available, only the LH and GT locks qualify.

Memory requirements fall into two categories: $O(LP)$ vs $O(L+P)$. One of the motivations for developing the MCS lock was indeed that it is $O(L + P)$, and the LH and M locks also achieve this.

Anderson's lock is clearly not competitive any more, requiring $O(LP)$ memory, a fetch-and-increment primitive, and more global operations than the other locks. The GT lock also requires $O(LP)$ memory, and it's advantage of requiring only an atomic swap primitive is equaled by the LH lock.

The MCS lock is competitive, but requires two global reads for hand-off compared to one read for the LH and M locks. With weaker consistency models, writes will typically be handled by a write buffer, thus global reads will dominate in large machines with fast processors.

This discussion makes evident that the M lock is either clearly superior or acceptable. It's drawbacks are more complex code and its requirement of a compare-and-clear primitive. It requires only $O(L + P)$ memory, and is either clearly faster or competitive in all critical paths of execution.

7 Conclusion

We have presented a detailed discussion of the performance, memory requirements, code size, and minimal hardware support for previously published queue locks, including two new locks.

The performance of the locks was studied by deducing the important critical paths and comparing the time required for all locks along these paths in terms of the primitive operations required.

We conclude that for a class of architectures, large cache-coherent shared-memory multiprocessors, the M lock is the overall preferred choice. If code requirements are tight and we do not wish the overhead of procedure calls, the LH lock provides good performance for smaller architectures and requires only an atomic swap.

References

[And90] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[AT92] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 1992 Real-Time Symposium, Phoenix, Arizona, December 2-4*, pages 12–21. IEEE, 1992.

[Cra93] T. S. Craig. Building FIFO and Priority-Queueing Spin Locks from Atomic Swap. Technical Report 93-02-02, Department of Computer Science and Engineering, FR-35, University of Washington, February 1993. Available via anonymous ftp from "cs.washington.edu" as "tr/1993/02/UW-CSE-93-02-02.PS.Z".

[Dig92] Digital Equipment Corp. *Alpha Architecture Handbook*, 1992.

[GT90] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[GVW89] J. R. Goodman, M. K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-scale Cache-Coherent Multiprocessors. In *Proceedings of the 3rd Architecture Symposium for Programming Languages and Operating Systems*, pages 64–75, 1989.

[Lam78] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[LL77] G. Le Lann. Distributed systems: towards a formal approach. In *IFIP Congress, North Holland*, pages 155–160, 1977.

[MCS91a] J.M. Mellor-Crummey and M.L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, pages 21–65, 1991.

[MCS91b] J.M. Mellor-Crummey and M.L. Scott. Synchronization Without Contention. In *Proceedings of the 4th Annual Architectural Support for Programming Languages and Operating Systems*, pages 269–278, 1991.

[MIP91] *MIPS R4000 Microprocessor User's Manual*, 1991.

[MLH94] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical Report T94:03, Swedish Institute of Computer Science, February 1994.

[PN85] G. F. Pfister and A. Norton. "Hot Spot" Contention and Combining in Multistage Interconenction Networks. *IEEE Transactions on Computers*, October 1985.

[SR84] Z. Segall and L. Rudolph. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.